

# **Programming Paradigms**

## **Lecture 8: Types (Part 1)**

**Prof. Dr. Michael Pradel**

**Software Lab, University of Stuttgart**

**Winter 2019/2020**

# Wake-up Exercise

---

What values do these JavaScript expressions evaluate to?

`''` == `'0'`  
`0` == `''`  
`0` == `'0'`  
`false` == `'false'`

# Wake-up Exercise

---

What values do these JavaScript expressions evaluate to?

```
' '    ==    '0'    // false
0      ==    ''     // true
0      ==    '0'    // true
false  ==    'false' // false
```

# Wake-up Exercise

---

What values do these JavaScript expressions evaluate to?

```
' '    ==    '0'    // false
0      ==    ''     // true
0      ==    '0'    // true
false  ==    'false' // false
```



**Two strings that are not the same**

# Wake-up Exercise

---

What values do these JavaScript expressions evaluate to?

```
' ' == '0' // false
0 == '' // true
0 == '0' // true
false == 'false' // false
```




**Number and string:  
String is coerced into a  
number (here: 0)**

# Wake-up Exercise

---

What values do these JavaScript expressions evaluate to?

```
''      == '0'      // false
0       == ''       // true
0       == '0'      // true
false  == 'false'  // false
```




**Boolean and another type:**

- Boolean gets coerced to a number (here: 0)
- String also get coerced to a number (here: NaN)
- The two numbers differ

# Overview

---

- **Introduction to Types and Type Systems** 
- **Type Checking**
  - Type Equivalence
  - Type Compatibility
  - Type Inference
- **Parametric Polymorphism**
- **Equality Testing and Assignments**

# Types

---

- **Most PLs: Expressions and memory objects have types**
- **Examples**
  - Assignment `x=4` (implicitly) says `x` has a number type
  - Declaration `int n;` says `n` has integer type
  - Expression `a+b` has a type, which depends on the type of `a` and `b`
  - `new X()` has a type



# Why Do We Need Types?

---

## Reason 1: **Provide context for operations**

- Meaning of  $a+b$  depends on types of  $a$  and  $b$ 
  - E.g., addition vs. string concatenation
- Meaning of `new x` depends in the type of  $x$ 
  - E.g., which initialization code to call?

**PL implementation uses this context information**

# Why Do We Need Types?

---

## Reason 2: **Limit valid operations**

- Many syntactically valid operations don't make any sense
  - Adding a character and a record
  - Computing the logarithm of a set

Helps **developers** find bugs early

# Why Do We Need Types?

---

## Reason 3: **Code readability and understandability**

- Types = stylized documentation
- Makes maintaining and extending code easier

**But: Sometimes, types make code **harder to write****

# Why Do We Need Types?

---

## Reason 4: **Compile-time optimizations**

- Compiler knows that some behavior is impossible
  - E.g., assignment of type T1 may not influence values of type T2

Works both for **explicitly specified** and **implicitly inferred types**

# Bits Are Untyped

---

- **(Most) hardware stores and computes on raw bits**
  - Bits may be code, integer data, addresses, etc.
- **(Most) assembly languages are untyped**
  - Operation of any kind can be applied to values at arbitrary locations

# Type Systems

---

- **Definition of types and their association with PL constructs**
  - Every PL construct that has/refers to a value has a type (e.g., named constants, variables, record fields, functions)
- **Rules for**
  - Type equivalence
  - Type compatibility
  - Type inference

# Type Checking

---

**Ensure that program obeys the type compatibility rules**

**Example (Java):**

```
int a = 3;  
String b = a - 2;
```

# Type Checking

---

**Ensure that program obeys the type compatibility rules**

**Example (Java):**

```
int a = 3;  
String b = a - 2;
```

**Type error: Can't assign int value to String variable**



# Strongly Typed PLs

---

**PL implementation enforces:**

**Operations only on values of proper type**

- Most PLs since 1970s
- C is mostly strongly typed
  - Exceptions, e.g.,:
    - Subroutines with variable number of parameters
    - Interoperability of pointers and arrays

# Statically Typed PLs

---

## Strongly typed and checked at compile-time

- Strictly speaking, practically no PL is statically typed
  - E.g., Java: Upcasts and reflection allow for runtime type errors
- In practice, means "mostly statically typed"

# Demo

---

**code/Casts.java: classes B and C extend A, List of As, add a B, cast to C; no compile-time but runtime error**

# Dynamically Typed PLs

---

## Type checking is delayed until runtime

- Type errors found only later in development process
- Common in “scripting languages”, e.g., JavaScript and Python
- Note: **Every value has a type** and type errors manifest as runtime errors

# Gradual Typing

---

## Middleground **between statically and dynamically typed PLs**

- Annotating types is optional
  - Can quickly write code and add types later
- Static type checker warns about errors obvious from the available types
  - **No guarantee to find all type errors**

# Example: Gradual Typing

---

**DEMO: code/gradual\_typing.py: function add\_numbers; pass int and str; show runtime error; run mypy; add first then second arg type**

# Quiz: \* Typed PLs

---

**What's the outcome of compiling and running this code in**

- a strongly typed language?
- a statically typed language?
- a dynamically typed language?

```
a = 23;  
b = true;  
c = a + a;  
d = c - b;  
print (d) ;
```

# Quiz: \* Typed PLs

---

What's the outcome of compiling and running this code in

- a strongly typed language?
- a statically typed language?
- a dynamically typed language?

```
a = 23;  
b = true;  
c = a + a;  
d = c - b;  
print (d);
```

**Strongly typed language:  
Type error** ←



# Quiz: \* Typed PLs


---

What's the outcome of compiling and running this code in

- a strongly typed language?
- a statically typed language?
- a dynamically typed language?

```
a = 23;  
b = true;  
c = a + a;  
d = c - b;  
print (d);
```

**Statically typed language:  
Compile-time type error**



# Quiz: \* Typed PLs

---

What's the outcome of compiling and running this code in

- a strongly typed language?
- a statically typed language?
- a dynamically typed language?

```
a = 23;  
b = true;  
c = a + a;  
d = c - b;  
print (d);
```

**Dynamically typed language:  
Runtime type error**



# Meaning of “Type”

---

## Three interpretations

- **Denotational**: Set of values
- **Structural**: Built-in, primitive type or composite type created from simpler types
- **Abstraction-based**: Interface that provides a set of operations

**In practice: Combination of all three**

# Polymorphism

---

- **Greek origin: “Having multiple forms”**
- **Two kinds**
  - **Parametric polymorphism:** Code takes (set of) type(s) as parameter
    - E.g., generics in Java, containers in C++
  - **Subtype polymorphism:** Extending or refining a supertype
    - E.g., subclasses in Java or C++

# Parametric Polymorphism

---

**DEMO: BinTree class with values of type T; setter; main method (try to set wrong type, then correct one)**

# Subtype Polymorphism

---

**DEMO: three classes (B and C extend A);  
method that takes an A; main method  
that passes a B**

# Polymorphic Variables

---

In some PLs, a **single variable** may refer to **objects of completely different types**

**Example (pseudo language):**

```
a = "abc"  
b = 42  
a = b  
a = "def"
```

# Polymorphic Variables

---

In some PLs, a **single variable** may refer to **objects of completely different types**

**Example (pseudo language):**

```
a = "abc"    // a holds a string
b = 42       // b holds an int
a = b        // a holds an int
a = "def"    // a holds a string (again)
```



# Polymorphic Variables

---

In some PLs, a **single variable** may refer to **objects of completely different types**

**Example (pseudo language):**

```
a = "abc"    // a holds a string
b = 42       // b holds an int
a = b        // a holds an int
a = "def"    // a holds a string (again)
```

**Type-correct in most dynamically typed  
(and even some statically typed) PLs**

# Special Types and Values

---

- **void type**: Indicates the absence of a type and has only one (trivial) value
- **null value**: Means “does not hold a value of its type”
- **Option types**: Indicates that the value may or may not hold a value of a specific type
  - E.g., `Option[int]` in Python means `int` or

None

# Quiz

---

**Which of the following statements is true?**

- A type system checks whether all types in a program are equivalent.
- PLs with dynamic scoping may be statically typed.
- Subclasses are a form of polymorphic typing.
- Option types cannot exist in strongly typed PLs.

# Quiz


---

**Which of the following statements is true?**

- ~~A type system checks whether all types in a program are equivalent.~~
- ~~PLs with dynamic scoping may be statically typed.~~
- Subclasses are a form of polymorphic typing.
- ~~Option types cannot exist in strongly typed PLs.~~

# Overview

---

- **Introduction to Types and Type Systems**
- **Type Systems**
  - Type Equivalence 
  - Type Compatibility
  - Type Inference
- **Parametric Polymorphism**
- **Equality Testing and Assignments**

# Type Equivalence

---

**Prerequisite for type checking:**

**Clarify **whether two types are equivalent****

**Two approaches**

- **Structural equivalence**
  - Same structure means same type
- **Name equivalence**
  - Same type name means same type

# Structural Equivalence

---

- Given two types, **compare** their **structure recursively**
- **Example: Any class with**
  - an int field called “age”,
  - a boolean field called “isRegistered”, and
  - a method called “printRecord”

# Variation Across Languages

---

- **Do names matter?**

- Same memory representation, but differently named
- E.g., different field names in a record

- **Does order matter?**

- Different memory representation, but lossless reordering possible
- E.g., same fields but in different order



(Pascal-like syntax)

T1 = record a: integer; b: real end;

T2 = record c: integer; d: real end;

T3 = record b: real; a: integer end;

T = record info: integer; next: ^T end;

U = record info: integer; next: ^V end;

V = record info: integer; next: ^U end;

# Limitation of Structural Equivalence

---

- Cannot distinguish **different concepts** that happen to be **represented the same way**
- **Example:**

```
type student = record
  name, address : string;
  age: integer
end;
```

vs.

```
type school = record
  name, address : string;
  age: integer
end;
```

# Limitation of Structural Equivalence

---

- Cannot distinguish **different concepts** that happen to be **represented the same way**
- **Example:**

```
type student = record
  name, address : string;
  age: integer
end;
```

VS.

```
type school = record
  name, address : string;
  age: integer
end;
```

```
{ This is allowed: }
x : student; y : school;
x := y;
```

# Name Equivalence

---

- Types with **different names** are **different**
- Assumption: Programmer wants it that way
- Used in many modern languages, e.g., Java

# Limitations of Name Equivalence

---

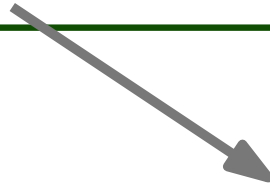
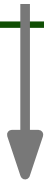
- **Alias types** cause difficulties
- **Example:**

```
{ Here, we want both types to be the same }  
type stack_element = integer;
```

```
{Here, we want distinct types,  
  to prevent mixed computations}  
type celsius = real;  
type fahrenheit = real;
```

# Strict vs. Loose Name Equivalence

---



- Aliases are **distinct types**
- `type A = B;` is a **definition**

- Aliases are **equivalent types**
- `type A = B;` is a **declaration**